(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 735 467 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
02.10.1996 Bulletin 1996/40

(51) Int. Cl.[6]: G06F 9/45

(21) Application number: 96104078.9

(22) Date of filing: 14.03.1996

(84) Designated Contracting States:
DE FR GB IT SE

(30) Priority: 29.03.1995 US 412546

(71) Applicant: SUN MICROSYSTEMS, INC.
Mountain View, CA 94043 (US)

(72) Inventors:
- Lim, Swee Boon
Mountain View, CA 94043 (US)
- Levy, Jacob Y.
Los Altos, CA 94022 (US)
- Kretsch, Donald J.
Cupertino, CA 95014 (US)
- Mitchell, Wesley E.
Sunnyvale, CA 94087 (US)
- Lerner, Benjamin
Palo Alto, CA 94303 (US)

(74) Representative: Liesegang, Roland, Dr.-Ing.
FORRESTER & BOEHMERT
Franz-Joseph-Strasse 38
80801 München (DE)

(54) Compiler with generic front end and dynamically loadable back ends

(57) A system and method provides for variable target outputs from a compiler with only a single execution. The compiler includes a front end, a generic back end, and plurality of individual back ends that are dynamically loaded by the compiler during execution. The front end produces an abstract syntax tree which is then processed by the generic back end and the individual back ends to produce a number of back end trees, each adapted for a specific back end, and representative of the desired structure of the various target outputs, including target code files, or events. The generic back end traverses the abstract syntax tree, and on each node of the tree, invokes each back end that has a node of its back end tree attached thereto. The back end can then modify its own back end tree, and attach further back end nodes to other nodes of the abstract syntax tree. In turn the generic back end will invoke such added attached nodes, until all attached back end nodes on all the nodes of the abstract syntax tree have been processed. This results in completed back end trees for all of the back ends, requiring only a single pass of the front end to produce the abstract syntax tree. The individual back ends then process their respective back end trees to produce their target outputs. A look up operation is provided in the generic back end that returns one or more back end nodes in response to a request therefore specifying a node of the abstract syntax tree, and a family identification value of a back end node.

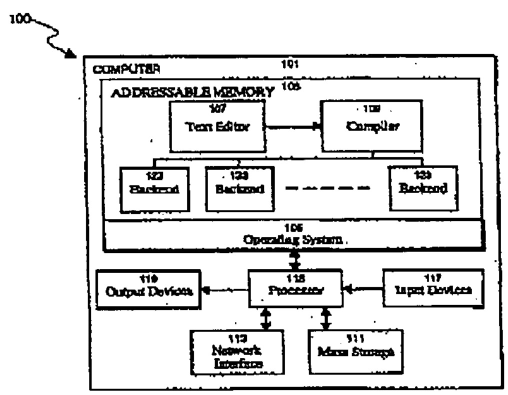

FIGURE 1

EP 0 735 467 A2

piler with dynamically loadable back ends. The system 100 includes a computer 101, having an addressable memory 103, a text editor 107, the compiler 109, and a plurality of independent back ends 123. The computer 101 is of conventional design, including a processor 115, input 117 and output devices 119, a network interface 113, and a mass storage device 111. The computer 101 may be realized by most general purposes computers, such as a SPARCstation™ computer manufactured by Sun Microsystems, Inc. of Mountain View, Ca. Any other general purpose computer may also be adapted for use with the invention. Computer 101 executes a general purpose operating system 105, such as Sun Microsystems' Solaris® operating system. The processor 115 executes the text editor 107, compiler 109, and back ends 123, including all of the operations thereof. The processor 115 also reads and writes source code files, and target output files to and from the mass storage device 111 during execution of the text editor 107 and compiler 109.

The text editor 107 is a conventional editor for creating and editing source code text files. The editor 107 provides conventional source code output as an ASCII or equivalent text file. The source code may be in any high level language, such as C, C++, SmallTalk, and the like.

Referring now to Figure 2, there is shown a dataflow diagram of the compiler 109 in cooperation with the back ends 123. The compiler 109 includes a front end 201 that receives the source code text files and creates therefrom in a conventional manner an abstract syntax tree. The abstract syntax tree is a graph representation of the syntax and structure of the source code text file. In a preferred embodiment, the compiler 109 operates on interface definition language (IDL) files that define object interfaces for distributed objects. In this preferred embodiment, the front end 201 produces an abstract syntax tree of an input IDL file or files.

In conjunction with the back ends 123, the compiler 109 may generate a variety of different target outputs. These target outputs include various header or code files, or actions upon other files or resources. For example, for a given input file x.idl, a preferred embodiment of the compiler 109 and back ends 123 may produce target output files such as x.hh, xsurrogates.hh, xsurrogates.cc, xmarshal.hh, xmarshal.cc, xtypes.hh, and xtypes.cc. Each of these target output files contain descriptions of data types, stubs and skeletons, and supporting marshaling routines, and is produced by one of the back ends 123. A back end 123 may produce any number of actions as target outputs, instead of producing files. For example, a back end 123 may load an interface repository with interface definitions derived from a given IDL file. To produce its particular output, each back end 123 usually requires more information than is available in the abstract syntax tree. Likewise, each back end 123 may use only certain information from the abstract syntax tree. For both of these reasons, each back end 123 creates its own back end tree that is particularly adapted to its target outputs.

The front end 201 provides the abstract syntax tree to a generic back end 203. The generic back end 203 coordinates the construction of the back end trees by the individual back ends 123. Each back end tree is derived from the abstract syntax tree, and is particularly adapted to an individual back end 123 for producing the target output from that back end 123. The discussion herein will refer to the abstract syntax tree and the individual back end trees, but it is understood that such "trees" are more generally defined as graphs, without the formal constraints of tree structures.

Figure 3a is a canonical illustration of a sample abstract syntax tree, herein abbreviated as "AST." The abstract syntax tree 300 is comprised of nodes, including the root AST node 301, and a number of child AST nodes 303. Each node in the abstract syntax tree 300 is an object with attributes that describe the source code token it represents, and operations for returning or modifying the node information. Both the root AST node 301 and the child AST nodes 303 will be jointly referred to as AST nodes 309 (not illustrated) when necessary. For ease of individual identification, the various nodes 309 are labeled T1 through T6 to denote their relative position in the abstract syntax tree 300. A given AST node may refer not just to its child nodes, but to any other AST node in the abstract syntax tree 300, for example, to provide a definition of a data type, or other useful information. For example, node T6 contains a reference to node T4.

Each back end 123 produces its own back end tree 500 in coordination with the generic back end 201. Figure 3f is an illustration of a sample back end tree derived from the abstract syntax tree 300. Each back end tree 500 includes a root back end node 501 and a number of child back end nodes 503. Both the root back end node 501 and the child back end nodes 503 will be jointly referred to as back end nodes 509 (not illustrated) when necessary. Each of the back end nodes 509 may correspond to one or more of the AST nodes 309, and accordingly, the structure of a back end tree 500 need not be a subset of the abstract syntax tree 300. This is illustrated in Figure 3d with node D6 depending from the root back end node D1 rather than node D5. In addition, each back end node 509 may be attached to one or more AST nodes 309; this form of attachment is shown in Figure 3d for node D5 which is attached to AST nodes T5 and T6. A back end node 509 may be directly attached to an AST node 309, or indirectly attached through an intermediate data structure. References to the back end tree 500 and back end nodes 509 in the remainder of this disclosure are intended to refer to any back end tree 500 and node 509 in general, and not to the specific illustrated trees shown in Figures 3a-f.

Referring again to Figure 2, in the preferred embodiment, there are three general types of back ends 123: an annotation back end 123a, an event generating back end 123b, and a code generating back end 123c. An

115 during the execution of the compiler 109 and generic back end 203.

The particular order in which the generic back end 203 attaches the back end nodes 509, including the order in which the root back end nodes 501 are attached 408, may be controlled to improve back end processing of the abstract syntax tree 300. In many cases, different back ends 123 share underlying data type information or other information that is useful to the production of their individual back end trees. For example, code generating back ends 123c for C and C++ would share common code for determining the sizes of data types in C or C++, or their type codes. Accordingly, it is desirable to have back end nodes 509 representing such common information available in the back end trees 500 prior to building or adding other back end nodes 509 that depend or reference such back end nodes or their common information.

In a preferred embodiment, the generic back end 203 satisfies this need by ordering the back end nodes 509 attached to each AST node 309 according to a priority level of the back end 123 associated with each back end node 509. In this preferred embodiment, the attach operation 205 attaches a specified back end node 509 according to a priority value of the back end 123 associated with the back end node 509. The priority level of the back ends 123 is preferably determined by a producer-consumer relationship between the different back ends 123 used in the system. Producer back ends, such as an annotation back end 123a, have higher priority than consumer back ends, such as a code generating back end 123c or event generating back end 123b. The particular priority ordering of any actual back end 123 depends on the other back ends 123 used in the system 100. The compiler developer can specify the priority level for each back end 123. In addition, priority can be established between back ends 123 of a given type. In alternate embodiments, priority schemes other than producer-consumer relationships, may be employed to determine a priority level of each back end 123.

Accordingly, the root back end nodes 501 are attached 408 to the root AST node 301 by the generic back end 203 according to the priority level of their respective back ends 123.

Once the abstract syntax tree 300 is populated 409, the generic back end 203 invokes a Do_Add operation on the root AST node 301. The Do_Add operation coordinates the individual back ends 123 to build their back end trees 500, each back end 123 processing selected AST nodes 309. The Do_Add operation calls each back end node 509 attached to each AST node 309, invoking an add operation 209 on each child AST node 303. The add operation 209 adds a back end node 503 to the back end tree 500 where the source token represented by the AST node 309 is relevant to the functionality of the back end 123. In some cases, the add operation 209 does not add a back end node 503 to the back end tree 500, because the underlying AST node 309 is not relevant to the target output of the back end 123. The Do_Add operation is preferably recursive, so that when the Do_Add is completed on the root AST node 301, the entire abstract syntax tree 300 has been traversed, and each back end tree 500 represents the output of the respective back end 123. The generic back end 201 then invokes 415 a Do_Produce operation, which in turn invokes the produce operation 211 of each back end 123 to produce the relevant target output files or actions for the back end 123.

Figure 4b illustrates one embodiment of the logic of the Do_Add operation. When the Do_Add operation is invoked, an AST node 309, here specified as "X", is passed in as the initial parameter. A pair of nested loops controls the processing in Do_Add. The outer loop iterates 419 over each child AST node Z of AST node X, and the inner loop iterates 421 over each back end node Y attached to the AST node X. This allows each such back end node to process each of the child AST nodes Z. These relationships of the various nodes are shown in Figure 4d. In Figure 4d, a portion of an abstract syntax 300 is shown, with an AST node 309, here node X, and any number of children AST nodes 303, here nodes Z1 through Zn. Attached to node X are any number of back end nodes 509, here labeled nodes Y1 through Yn.

In the core of the loops, the add operation 209 of a back end node Y is invoked 423 on the child AST node Z. In the example of Figure 4d, the add operation 209 of Y1 would be invoked on Z1, and then Y2 on Z1, and so on, through Yn. The add operation 209 determines whether the child AST node Z is relevant to the back end tree 500 of the back end 123. This determination may be based on the data type of the child AST node Z or its parent AST node X, or more generally, on the desired structure of the back end tree 500. For example, the add operation 209 of a back end node 509 for a code generating back end 123c, would not add a back end node for a AST node Z that represents a token from an included source code file. If the child AST node Z is relevant to the back end 123 functionality, then one or more new back end node(s) 509 are created and added to the back end tree 500. In addition, the add operation 209 may invoke the attach operation 205 on any number of AST nodes 309, to attach the newly created back end node(s) 509 to such AST node(s) 309.

As indicated, the add operation 209 includes two logical, though not necessarily actual, parameters. A first parameter specifies the back end node Y in the back end tree 500 associated with the AST node X on which the add operation 209 is to be performed, and a second parameter specifies a child AST node Z of the AST node X. In an object oriented embodiment, the add operation 209 is included in each of nodes of the back end tree 500. For example, in C++, the invocation may be Y->add(Z) where "Y" and "Z" are pointers to nodes in the respective trees. In a procedural embodiment, the add operation 209 is part of the procedural code for each back end 123. In either case, each back end 123 controls the functionality of the add operation 209. In a

tree a target output, the back end tree having at least one back end tree node; and,

a generic back end that controls the at least one back end to produce the associated back end tree, such that each back end is independent of the generic back end, and dynamically loaded by the apparatus during execution.

2. The apparatus of claim 1 wherein the generic back end invokes in a selected order each back end associated with each back end node attached to each abstract syntax tree node.

3. The apparatus of claim 1 or 2, wherein each back end further comprises:

an add operation, invocable by the generic back end, that adds zero or more back end nodes to a back end tree.

4. The apparatus of claim 3, wherein the generic back end further comprises:

an attach operation, invocable by a back end, that attaches a back end node of a back end tree to at least one node of the abstract syntax tree.

5. The apparatus of claim 4, wherein each back end node is attached to the abstract syntax tree according to a priority value corresponding to a priority value of the back end associated with the back end node.

6. The apparatus of claim 5, wherein selected back end nodes associated with one back end are attached to the abstract syntax tree in a predetermined order.

7. The apparatus of claim 3, wherein each back end node is an object and the add operation is a method of the object.

8. The apparatus of one of claims 1 to 7 further comprising:

a memory device that stores the abstract syntax tree, the at least one back end tree, the generic back end, and the at least one back end; and

a processing device that executes the generic back end and the at least one back end to produce from the at least one back end at least one target output.

9. The apparatus of one of claims 1 to 8, wherein each back end and each back end node associated with the back end has a family identification value, wherein the generic back end further comprises:

a look up operation that returns zero or more nodes from at least one back end tree having a family identification value and attached to a node of the abstract syntax tree in response to a request from a back end specifying the family identification value and the node of the abstract syntax tree.

10. A method for compiling a source code file and producing variable target outputs thereform, comprising the steps of:

providing a source code file;

executing a compiler having a front end and a generic back end;

loading at least one back end to operate with the generic back end;

creating an abstract syntax tree from the source code with the front end, the abstract syntax tree including a root node and at least one child node;

generating a back end tree associated with each back end; and

producing from each back end tree a target output.

11. The method of claim 10, wherein the step of generating a back end tree is performed by selectively invoking each back end on selected nodes of the abstract syntax tree.

12. The method of claim 10 or 11, further comprising the steps of:

attaching to the root node of the abstract syntax tree a back end node of at least one back end tree;

selecting nodes of the abstract syntax tree, and for each selected node, adding to selected back end trees zero or more back end nodes representing syntactic information derived from the abstract syntax tree; and

attaching selected back end nodes to selected nodes of the abstract syntax tree.

13. The method of claim 12, further comprising the step of:

repeating the selecting step on each back end node attached to each node of the abstract syntax tree.

14. The method of claim 12, wherein each attaching step further comprises the step of:

attaching the back end node to the abstract syntax tree according to a priority value corresponding to a priority value of the back end associated with the back end node.

15. The method of claim 14, further comprising the step of:

attaching selected back end nodes associ-

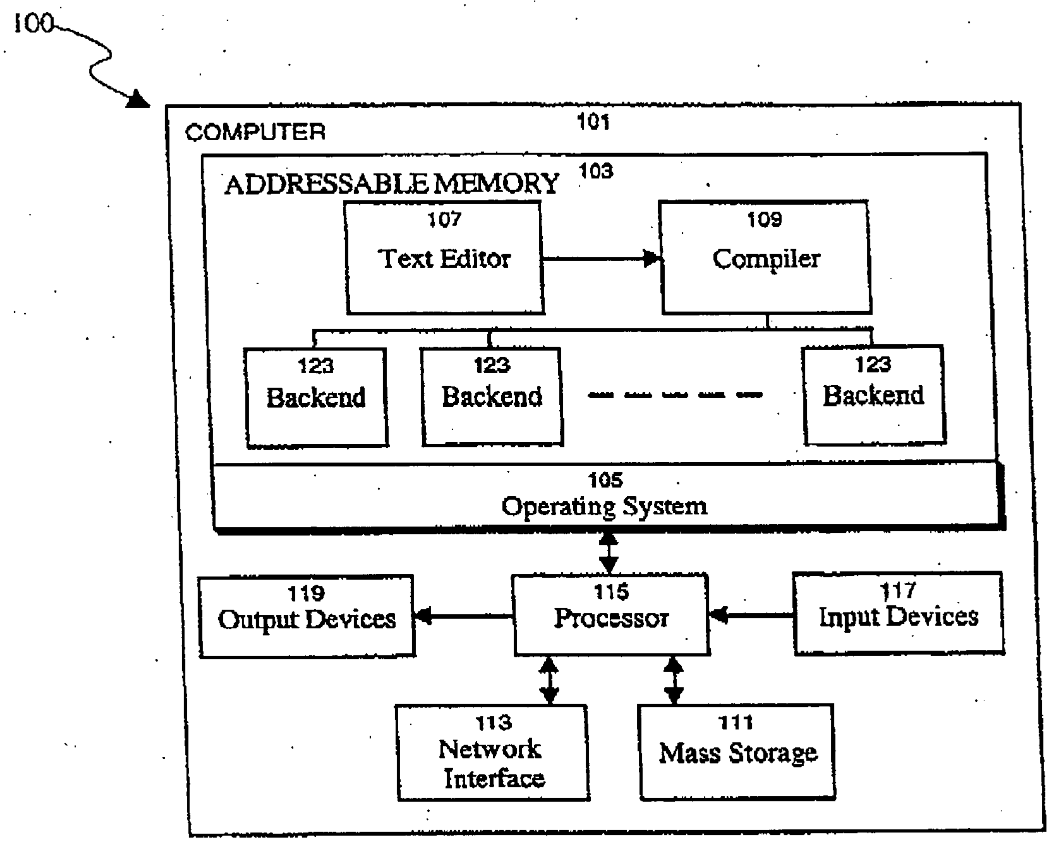
100
COMPUTER
101
ADDRESSABLE MEMORY
103
107
Text Editor
109
Compiler
123
Backend
123
Backend
123
Backend
105
Operating System
119
Output Devices
115
Processor
117
Input Devices
113
Network Interface
111
Mass Storage

FIGURE 1

FIGURE 3a

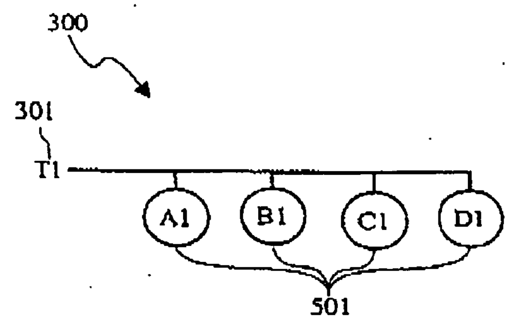

FIGURE 3b

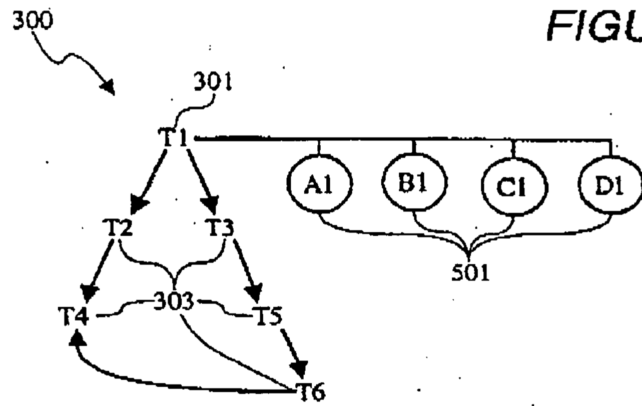

FIGURE 3c

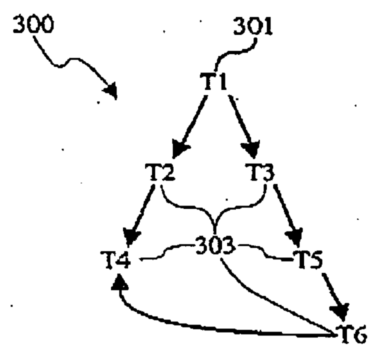

FIGURE 3e

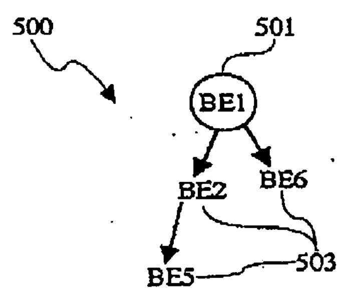

FIGURE 3f

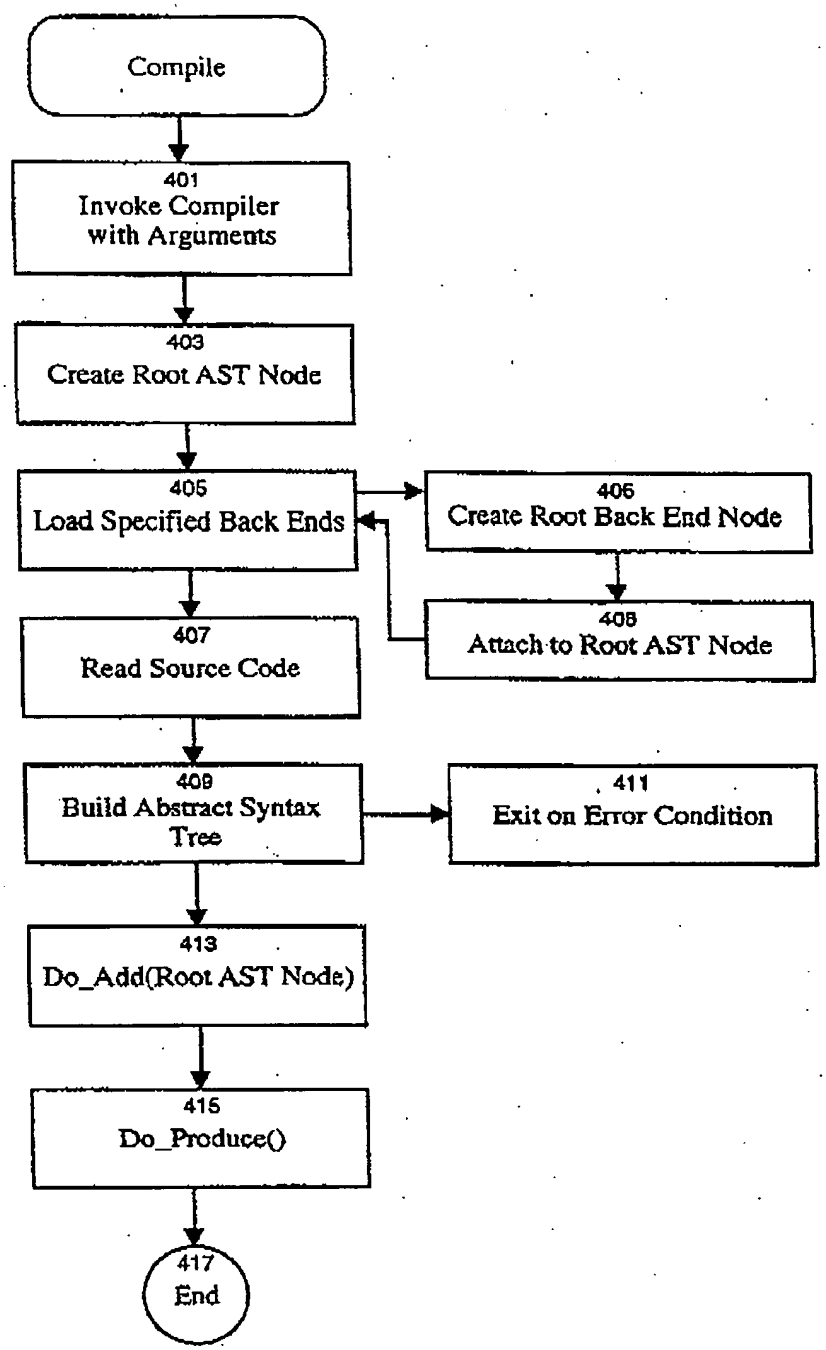
Compile
401
Invoke Compiler with Arguments
403
Create Root AST Node
405
Load Specified Back Ends
406
Create Root Back End Node
408
Attach to Root AST Node
407
Read Source Code
409
Build Abstract Syntax Tree
411
Exit on Error Condition
413
Do_Add(Root AST Node)
415
Do_Produce()
417
End

FIGURE 4a

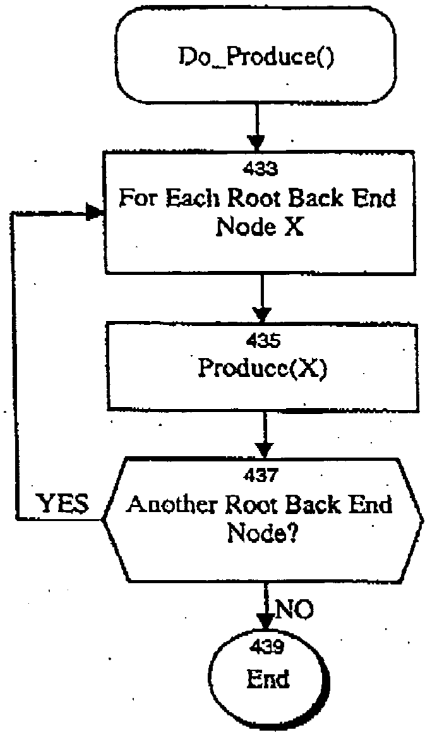

FIGURE 4c

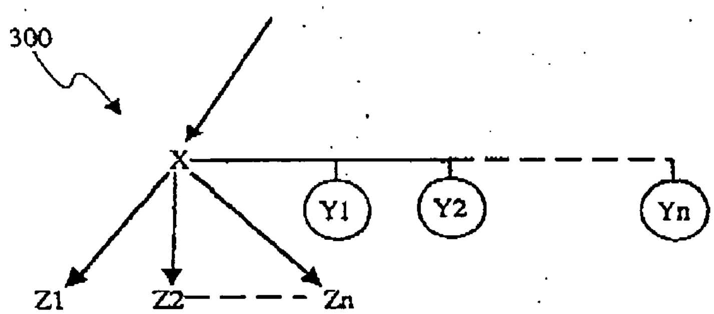

FIGURE 4d

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 735 467 A3

(12) EUROPEAN PATENT APPLICATION

(88) Date of publication A3:
02.05.1997 Bulletin 1997/18

(43) Date of publication A2:
02.10.1996 Bulletin 1996/40

(21) Application number: 96104078.9

(22) Date of filing: 14.03.1996

(51) Int. Cl.⁶: G06F 9/45

(84) Designated Contracting States:
DE FR GB IT SE

(30) Priority: 29.03.1995 US 412546

(71) Applicant: SUN MICROSYSTEMS, INC.
Mountain View, CA 94043 (US)

(72) Inventors:
- Lim, Swee Boon
  Mountain View, CA 94043 (US)
- Levy, Jacob Y.
  Los Altos, CA 94022 (US)
- Kretsch, Donald J.
  Cupertino, CA 95014 (US)
- Mitchell, Wesley E.
  Sunnyvale, CA 94087 (US)
- Lerner, Benjamin
  Palo Alto, CA 94303 (US)

(74) Representative: Liesegang, Roland, Dr.-Ing.
FORRESTER & BOEHMERT
Franz-Joseph-Strasse 38
80801 München (DE)

(54) Compiler with generic front end and dynamically loadable back ends

(57) A system and method provides for variable target outputs from a compiler with only a single execution. The compiler includes a front end, a generic back end, and plurality of individual back ends that are dynamically loaded by the compiler during execution. The front end produces an abstract syntax tree which is then processed by the generic back end and the individual back ends to produce a number of back end trees, each adapted for a specific back end, and representative of the desired structure of the various target outputs, including target code files, or events. The generic back end traverses the abstract syntax tree, and on each node of the tree, invokes each back end that has a node of its back end tree attached thereto. The back end can then modify its own back end tree, and attach further back end nodes to other nodes of the abstract syntax tree. In turn the generic back end will invoke such added attached nodes, until all attached back end nodes on all the nodes of the abstract syntax tree have been processed. This results in completed back end trees for all of the back ends, requiring only a single pass of the front end to produce the abstract syntax tree. The individual back ends then process their respective back end trees to produce their target outputs. A look up operation is provided in the generic back end that returns one or more back end nodes in response to a request therefore specifying a node of the abstract syntax tree, and a family identification value of a back end node.

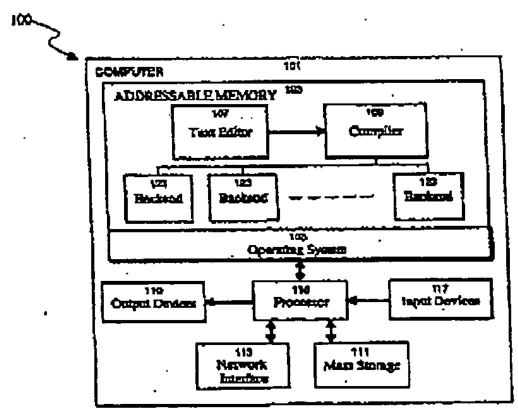

FIGURE 1

EP 0 735 467 A3

Printed by Rank Xerox (UK) Business Services
2.14.4/3.4